



Science and
Technology
Facilities Council



GALAHAD

SLLS

USER DOCUMENTATION

GALAHAD Optimization Library version 5.1

1 SUMMARY

This package uses a preconditioned, projected-gradient method to solve the **simplex-constrained regularized linear least-squares problem**

$$\text{minimize } q(\mathbf{x}) = \frac{1}{2} \|\mathbf{A}_o \mathbf{x} - \mathbf{b}\|_{\mathbf{W}}^2 + \frac{1}{2} \boldsymbol{\sigma} \|\mathbf{x}\|_2^2,$$

where the variables \mathbf{x} are required to lie within the **regular simplex**

$$\mathbf{e}^T \mathbf{x} = 1 \text{ and } \mathbf{x} \geq 0,$$

where the o by n real matrix \mathbf{A}_o , the vector \mathbf{b} and the non-negative weights \mathbf{w} and $\boldsymbol{\sigma}$ are given, where \mathbf{e} is the vector of ones, and where the Euclidean and weighted-Euclidean norms are given by $\|\mathbf{v}\|_2^2 = \mathbf{v}^T \mathbf{v}$ and $\|\mathbf{v}\|_{\mathbf{W}}^2 = \mathbf{v}^T \mathbf{W} \mathbf{v}$, respectively, with $\mathbf{W} = \text{diag}(\mathbf{w})$. Full advantage is taken of any zero coefficients of the Jacobian matrix \mathbf{A}_o of the **residuals** $\mathbf{r}(\mathbf{x}) = \mathbf{A}_o \mathbf{x} - \mathbf{b}$; the matrix need not be provided as there are options to obtain matrix-vector products involving \mathbf{A}_o and its transpose either by reverse communication or from a user-provided subroutine.

ATTRIBUTES — Versions: GALAHAD_SLLS_single, GALAHAD_SLLS_double. **Uses:** GALAHAD_CPU_time, GALAHAD_SYMBOLS, GALAHAD_SPACE, GALAHAD_STRING, GALAHAD_SORT, GALAHAD_NORMS, GALAHAD_CONVERT, GALAHAD_SBLS, GALAHAD_QPT, GALAHAD_QPD, GALAHAD_USERDATA, GALAHAD_SPECFILE. **Date:** July 2022. **Origin:** N. I. M. Gould, STFC-Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003.

2 HOW TO USE THE PACKAGE

The package is available with single, double and (if available) quadruple precision reals, and either 32-bit or 64-bit integers. Access to the 32-bit integer, single precision version requires the `USE` statement

```
USE GALAHAD_SLLS_single
```

with the obvious substitution `GALAHAD_SLLS_double`, `GALAHAD_SLLS_quadruple`, `GALAHAD_SLLS_single_64`, `GALAHAD_SLLS_double_64` and `GALAHAD_SLLS_quadruple_64` for the other variants.

If it is required to use more than one of the modules at the same time, the derived types `SMT_type`, `QPT_problem_type`, `NLPT_userdata_type`, `SLLS_time_type`, `SLLS_control_type`, `SLLS_inform_type` and `SLLS_data_type` (Section 2.3) and the subroutines `SLLS_initialize`, `SLLS_solve`, `SLLS_terminate`, (Section 2.4) and `SLLS_read_specfile` (Section 2.8) must be renamed on one of the `USE` statements.

2.1 Matrix storage formats

When it is explicitly available, an m by n matrix \mathbf{A} (in our case \mathbf{A}_o) may be stored in a variety of input formats.

2.1.1 Dense row-wise storage format

The matrix \mathbf{A} is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. Component $n * (i - 1) + j$ of the storage array `A%val` will hold the value $\mathbf{A}_{i,j}$ for $i = 1, \dots, m$, $j = 1, \dots, n$.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

2.1.2 Dense column-wise storage format

The matrix \mathbf{A} is stored as a compact dense matrix by columns, that is, the values of the entries of each column in turn are stored in order within an appropriate real one-dimensional array. Component $m * (j - 1) + i$ of the storage array `A%val` will hold the value $\mathbf{A}_{i,j}$ for $i = 1, \dots, m, j = 1, \dots, n$.

2.1.3 Sparse coordinate storage format

Only the nonzero entries of the matrices are stored. For the l -th entry of \mathbf{A} , its row index i , column index j and value \mathbf{A}_{ij} are stored in the l -th components of the integer arrays `A%row`, `A%col` and real array `A%val`. The order is unimportant, but the total number of entries `A%ne` is required.

2.1.4 Sparse row-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in row i appear directly before those in row $i + 1$. For the i -th row of \mathbf{A} , the i -th component of the integer array `A%ptr` holds the position of the first entry in this row, while `A%ptr(m + 1)` holds the total number of entries plus one. The column indices j and values \mathbf{A}_{ij} of the entries in the i -th row are stored in components $l = \text{A\%ptr}(i), \dots, \text{A\%ptr}(i + 1) - 1$ of the integer array `A%col`, and real array `A%val`, respectively.

2.1.5 Sparse column-wise storage format

Yet again only the nonzero entries are stored, but this time they are ordered so that those in column j appear directly before those in column $j + 1$. For the j -th column of \mathbf{A} , the j -th component of the integer array `A%ptr` holds the position of the first entry in this column, while `A%ptr(n + 1)` holds the total number of entries plus one. The row indices i and values \mathbf{A}_{ij} of the entries in the j -th column are stored in components $l = \text{A\%ptr}(j), \dots, \text{A\%ptr}(j + 1) - 1$ of the integer array `A%row`, and real array `A%val`, respectively.

For sparse matrices, the row- and column-wise storage schemes almost always requires less storage than their predecessor.

2.2 Real and integer kinds

We use the terms integer and real to refer to the fortran keywords `REAL(rp_)` and `INTEGER(ip_)`, where `rp_` and `ip_` are the relevant kind values for the real and integer types employed by the particular module in use. The former are equivalent to default `REAL` for the single precision versions, `DOUBLE PRECISION` for the double precision cases and quadruple-precision if 128-bit reals are available, and correspond to `rp_ = real32`, `rp_ = real64` and `rp_ = real128` respectively as defined by the fortran `iso_fortran_env` module. The latter are default (32-bit) and long (64-bit) integers, and correspond to `ip_ = int32` and `ip_ = int64`, respectively, again from the `iso_fortran_env` module.

2.3 The derived data types

Ten derived data types are accessible from the package.

2.3.1 The derived data type for holding matrices

The derived data type `SMT_TYPE` is used to hold the matrix \mathbf{A} . The components of `SMT_TYPE` used here are:

- `m` is a scalar component of type `INTEGER(ip_)`, that holds the number of rows in the matrix.
- `n` is a scalar component of type `INTEGER(ip_)`, that holds the number of columns in the matrix.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

- `ne` is a scalar variable of type `INTEGER(ip_)`, that holds the number of matrix entries.
- `type` is a rank-one allocatable array of type default `CHARACTER`, that is used to indicate the matrix storage scheme used. Its precise length and content depends on the type of matrix to be stored (see §2.3.2).
- `val` is a rank-one allocatable array of type `REAL(rp_)` and dimension at least `ne`, that holds the values of the entries. Any duplicated entries that appear in the sparse coordinate or row-wise schemes will be summed.
- `row` is a rank-one allocatable array of type `INTEGER(ip_)`, and dimension at least `ne`, that may hold the row indices of the entries. (see §2.1.3 and §2.1.5).
- `col` is a rank-one allocatable array of type `INTEGER(ip_)`, and dimension at least `ne`, that may hold the column indices of the entries (see §2.1.3–2.1.4).
- `ptr` is a rank-one allocatable array of type `INTEGER(ip_)`, and dimension at least `m + 1`, that may hold the pointers to the first entry in each row (see §2.1.4). or dimension at least `n + 1`, that may hold the pointers to the first entry in each column (see §2.1.5).

2.3.2 The derived data type for holding the problem

The derived data type `QPT_problem_type` is used to hold the problem. The components of `QPT_problem_type` are:

- `n` is a scalar variable of type `INTEGER(ip_)`, that holds the number of optimization variables, n .
- `o` is a scalar variable of type `INTEGER(ip_)`, that holds the number of residuals, o .
- `Ao` is scalar variable of type `SMT_TYPE` that holds the design matrix \mathbf{A}_o (if it is available). The following components are used here:

`Ao%type` is an allocatable array of rank one and type default `CHARACTER`, that is used to indicate the storage scheme used. If the dense row-wise storage scheme (see Section 2.1.1) is used, the first twelve components of `Ao%type` must contain the string `DENSE_BY_ROWS`, while if the column-wise scheme (see Section 2.1.2) is used, the fifteen components of `Ao%type` must contain the string `DENSE_BY_COLUMNS`. For the sparse coordinate scheme (see Section 2.1.3), the first ten components of `Ao%type` must contain the string `COORDINATE`, for the sparse row-wise storage scheme (see Section 2.1.4), the first fourteen components of `Ao%type` must contain the string `SPARSE_BY_ROWS`. and for the sparse column-wise storage scheme (see Section 2.1.5), the first seventeen components of `Ao%type` must contain the string `SPARSE_BY_COLUMNS`.

For convenience, the procedure `SMT_put` may be used to allocate sufficient space and insert the required keyword into `Ao%type`. For example, if `nlp` is of derived type `SLLS_problem_type` and involves a design matrix we wish to store using the coordinate scheme, we may simply

```
CALL SMT_put( nlp%A%type, 'COORDINATE' )
```

See the documentation for the GALAHAD package `SMT` for further details on the use of `SMT_put`.

`Ao%type` should **not** be allocated if \mathbf{A}_o is unavailable (and access provided to \mathbf{A}_o provided by other means, see later sections), and in this case the remaining components of `A` need not be set.

`Ao%ne` is a scalar variable of type `INTEGER(ip_)`, that holds the number of entries in \mathbf{A}_o in the sparse coordinate storage scheme (see Section 2.1.3). It need not be set for any of the other three schemes.

`Ao%val` is a rank-one allocatable array of type `REAL(rp_)`, that holds the values of the entries of the design matrix \mathbf{A}_o in any of the storage schemes discussed in Section 2.1.

`Ao%row` is a rank-one allocatable array of type `INTEGER(ip_)`, that holds the row indices of \mathbf{A}_o in the sparse coordinate storage scheme (see Section 2.1.3). It need not be allocated for any of the other three schemes.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

- `Ao%col` is a rank-one allocatable array variable of type `INTEGER(ip_)`, that holds the column indices of \mathbf{A}_o in either the sparse coordinate (see Section 2.1.3), or the sparse row-wise (see Section 2.1.4) storage scheme. It need not be allocated when the dense or column-wise storage schemes are used.
- `Ao%row` is a rank-one allocatable array variable of type `INTEGER(ip_)`, that holds the row indices of \mathbf{A}_o in either the sparse coordinate (see Section 2.1.3), or the sparse column-wise (see Section 2.1.5) storage scheme. It need not be allocated when the dense or row-wise storage schemes are used.
- `Ao%ptr` is a rank-one allocatable array and type `INTEGER(ip_)`, that must be of dimension $o+1$ and hold the starting position of each row of \mathbf{A}_o , as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.1.4). If the sparse column-wise storage scheme (see Section 2.1.5) is used, it must instead be of dimension $n+1$ and hold the starting position of each column of \mathbf{A}_o , as well as the total number of entries plus one. It need not be allocated when the other schemes are used.
- `B` is a rank-one allocatable array of dimension o and type `REAL(rp_)`, that holds the constant term \mathbf{b} in the residuals. The i -th component of `B`, $i = 1, \dots, o$, contains \mathbf{b}_i .
- `R` is a rank-one allocatable array of dimension o and type `REAL(rp_)`, that holds the residuals, $\mathbf{r}(\mathbf{x}) = \mathbf{A}_o \mathbf{x} - \mathbf{b}$, at the point \mathbf{x} . The i -th component of `R`, $i = 1, \dots, o$, contains $\mathbf{r}_i(\mathbf{x})$.
- `G` is a rank-one allocatable array of dimension n and type `REAL(rp_)`, that holds the gradient of the objective $\mathbf{g}(\mathbf{x}) = \mathbf{A}_o^T \mathbf{W} \mathbf{r}(\mathbf{x})$, at the point \mathbf{x} . The j -th component of `G`, $j = 1, \dots, n$, contains $\mathbf{g}_j(\mathbf{x})$.
- `X` is a rank-one allocatable array of dimension n and type `REAL(rp_)`, that holds the values \mathbf{x} of the optimization variables. The j -th component of `X`, $j = 1, \dots, n$, contains x_j .
- `Z` is a rank-one allocatable array of dimension n and type default `REAL(rp_)`, that holds the values \mathbf{z} of estimates of the dual variables corresponding to the non-negativity constraints (see Section 4). The j -th component of `Z`, $j = 1, \dots, n$, contains z_j .

2.3.3 The derived data type for holding control parameters

The derived data type `SLLS_control_type` is used to hold controlling data. Default values may be obtained by calling `SLLS_initialize` (see Section 2.4.1), while components may also be changed by calling `SLLS_read_specfile` (see Section 2.8.1). The components of `SLLS_control_type` are:

- `error` is a scalar variable of type `INTEGER(ip_)`, that holds the stream number for error messages. Printing of error messages in `SLLS_solve` and `SLLS_terminate` is suppressed if `error` ≤ 0 . The default is `error` = 6.
- `out` is a scalar variable of type `INTEGER(ip_)`, that holds the stream number for informational messages. Printing of informational messages in `SLLS_solve` is suppressed if `out` < 0 . The default is `out` = 6.
- `print_level` is a scalar variable of type `INTEGER(ip_)`, that is used to control the amount of informational output which is required. No informational output will occur if `print_level` ≤ 0 . If `print_level` = 1, a single line of output will be produced for each iteration of the process. If `print_level` ≥ 2 , this output will be increased to provide significant detail of each iteration. The default is `print_level` = 0.
- `start_print` is a scalar variable of type `INTEGER(ip_)`, that specifies the first iteration for which printing will be permitted in `GALAHAD_SLLS_solve`. If `start_print` is negative, printing will be permitted from the outset. The default is `start_print` = -1.
- `stop_print` is a scalar variable of type `INTEGER(ip_)`, that specifies the last iteration for which printing will be permitted in `GALAHAD_SLLS_solve`. If `stop_print` is negative, printing will be permitted once it has been started by `start_print`. The default is `stop_print` = -1.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

`print_gap` is a scalar variable of type `INTEGER(ip_)`. Once printing has been started, output will occur once every `print_gap` iterations. If `print_gap` is no larger than 1, printing will be permitted on every iteration. The default is `print_gap = 1`.

`maxit` is a scalar variable of type `INTEGER(ip_)`, that holds the maximum number of iterations which will be allowed in `GALAHAD_SLLS_solve`. The default is `maxit = 1000`.

`cold_start` is a scalar variable of type `INTEGER(ip_)`, that should be set to 0 if a warm start is required (with variables assigned according to `X_stat`, see below), and to any other value if the values given in `prob%X` suffice. The default is `cold_start = 1`.

`preconditioner` is a scalar variable of type `INTEGER(ip_)`, that specifies the type of preconditioner (scaling) used when computing the search direction. Currently this can be 0 (no preconditioner), 1 (a diagonal preconditioner that normalises the rows of \mathbf{A}_o) or 2 (a preconditioner supplied by the user either via a subroutine call of `eval_PREC`, see §2.5.4, or via reverse communication, see §2.6), although other values may be introduced in future. The default is `preconditioner = 1`.

`change_max` is a scalar variable of type `INTEGER(ip_)`, that specifies the maximum number of per-iteration changes in the working set permitted when allowing subspace solution rather than steepest descent (see §4). The default is `change_max = 2`.

`cg_maxit` is a scalar variable of type `INTEGER(ip_)`, that holds the maximum number of conjugate-gradient iterations which will be allowed per main iteration in `GALAHAD_SLLS_solve`. The default is `cg_maxit = 1000`, and any negative value will be interpreted as $n + 1$.

`arcsearch_max_steps` is a scalar variable of type `INTEGER(ip_)`, that specifies the maximum number of steps allowed in an individual piecewise arc search. The default is `arcsearch_max_steps = 1000`, and a negative value removes the limit.

`weight` is a scalar variable of type `REAL(rp_)`, that is used to specify the weight σ that controls regularization of the objective function. Any value smaller than 0 will be regarded as zero. The default is `weight = 0.0`.

`stop_d` is a scalar variable of type default `REAL(rp_)`, that holds the required accuracy for the dual infeasibility (see Section 4). The default is `stop_d = $u^{1/3}$` , where u is `EPSILON(1.0)` (`EPSILON(1.0D0)` in `GALAHAD_SLLS_double`).

`stop_cg_relative` and `stop_cg_absolute` are scalar variables of type `REAL(rp_)`, that hold the relative and absolute convergence tolerances for the conjugate-gradient iteration that occurs in the face of currently-active constraints when constructing the search direction. `stop_cg_relative = 0.01` and `stop_cg_absolute = \sqrt{u}` , where u is `EPSILON(1.0)` (`EPSILON(1.0D0)` in `GALAHAD_SLLS_double`).

`alpha_max` is a scalar variable of type `REAL(rp_)`, that specifies The default is `alpha_max = 10^{20}` .

`alpha_initial` is a scalar variable of type `REAL(rp_)`, that specifies the initial arc length during the inexact piecewise arc search. The default is `alpha_initial = 1.0`.

`alpha_reduction` is a scalar variable of type `REAL(rp_)`, that specifies the arc length reduction factor for the inexact piecewise arc search. The default is `alpha_reduction = 0.5`.

`arcsearch_acceptance_tol` is a scalar variable of type `REAL(rp_)`, that specifies the required relative reduction during the inexact arc search The default is `arcsearch_acceptance_tol = 0.01`.

`stabilisation_weight` is a scalar variable of type `REAL(rp_)`, that specifies the weight added to the search-direction subproblem to stabilise the computation. The default is `stabilisation_weight = 10^{-12}` .

`cpu_time_limit` is a scalar variable of type `REAL(rp_)`, that is used to specify the maximum permitted CPU time. Any negative value indicates no limit will be imposed. The default is `cpu_time_limit = - 1.0`.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

`direct_subproblem_solve` is a scalar variable of type default LOGICAL, that should be set `.TRUE.` if the search over a promising subspace is carried out using matrix factorization, and `.FALSE.` if the conjugate-gradient least-squares (CGLS) method is to be preferred. The former is generally more effective so long as the cost of factorization isn't exorbitant. The default is `direct_subproblem_solve = .TRUE.`, but the package will override this if the Jacobian is not explicitly available.

`exact_arc_search` is a scalar variable of type default LOGICAL, that should be set `.TRUE.` if the exact minimizer along the search arc is required, and `.FALSE.` if an approximation found by backtracking or advancing suffices. The default is `exact_arc_search = .TRUE.`.

`space_critical` is a scalar variable of type default LOGICAL, that must be set `.TRUE.` if space is critical when allocating arrays and `.FALSE.` otherwise. The package may run faster if `space_critical` is `.FALSE.` but at the possible expense of a larger storage requirement. The default is `space_critical = .FALSE.`.

`deallocate_error_fatal` is a scalar variable of type default LOGICAL, that must be set `.TRUE.` if the user wishes to terminate execution if a deallocation fails, and `.FALSE.` if an attempt to continue will be made. The default is `deallocate_error_fatal = .FALSE.`.

`prefix` is a scalar variable of type default CHARACTER and length 30, that may be used to provide a user-selected character string to preface every line of printed output. Specifically, each line of output will be prefaced by the string `prefix(2:LEN(TRIM(prefix))-1)`, thus ignoring the first and last non-null components of the supplied string. If the user does not want to preface lines by such a string, they may use the default `prefix = ""`.

`SBLS_control` is a scalar variable of type `SBLS_control_type` whose components are used to control the factorization and/or preconditioner used, performed by the package `GALAHAD_SBLS`. See the documentation for `GALAHAD_SBLS` for further details.

2.3.4 The derived data type for holding timing information

The derived data type `SLLS_time_type` is used to hold elapsed CPU times for the various parts of the calculation. The components of `SLLS_time_type` are:

`total` is a scalar variable of type default REAL, that gives the total time spent in the package.

`analyse` is a scalar variable of type default REAL, that gives the time spent analysing the required matrices prior to factorization.

`factorize` is a scalar variable of type default REAL, that gives the time spent factorizing the required matrices.

`solve` is a scalar variable of type default REAL, that gives the time spent computing the search direction.

2.3.5 The derived data type for holding informational parameters

The derived data type `SLLS_inform_type` is used to hold parameters that give information about the progress and needs of the algorithm. The components of `SLLS_inform_type` are:

`status` is a scalar variable of type `INTEGER(ip_)`, that gives the exit status of the algorithm. See Section 2.7 for details.

`alloc_status` is a scalar variable of type `INTEGER(ip_)`, that gives the status of the last attempted array allocation or deallocation. This will be 0 if `status = 0`.

`bad_alloc` is a scalar variable of type default CHARACTER and length 80, that gives the name of the last internal array for which there were allocation or deallocation errors. This will be the null string if `status = 0`.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

`factorization_status` is a scalar variable of type `INTEGER(ip_)`, that gives the return status from the matrix factorization.

`iter` is a scalar variable of type `INTEGER(ip_)`, that gives the number of iterations performed.

`obj` is a scalar variable of type `REAL(rp_)`, that holds the value of the objective function at the best estimate of the solution found.

`time` is a scalar variable of type `SLLS_time_type` whose components are used to hold elapsed CPU times for the various parts of the calculation (see Section 2.3.4).

`SBLs_inform` is a scalar variable of type `SBLs_inform_type` whose components provide information about the progress and needs of the factorization/preconditioner performed by the package `GALAHAD_SBLs`. See the documentation for `GALAHAD_SBLs` for further details.

2.3.6 The derived data type for holding problem data

The derived data type `SLLS_data_type` is used to hold all the data for a particular problem, or sequences of problems with the same structure, between calls of `SLLS` procedures. This data should be preserved, untouched, from the initial call to `SLLS_initialize` to the final call to `SLLS_terminate`.

2.3.7 The derived data type for holding user data

The derived data type `NLPT_userdata_type` is available to allow the user to pass data to and from user-supplied matrix-vector product and preconditioning subroutines (see Section 2.5). Components of variables of type `NLPT_userdata_type` may be allocated as necessary. The following components are available:

`integer` is a rank-one allocatable array of type `INTEGER(ip_)`.

`real` is a rank-one allocatable array of type default `REAL(rp_)`

`complex` is a rank-one allocatable array of type default `COMPLEX` (double precision complex in `GALAHAD_SLLS_-double`).

`character` is a rank-one allocatable array of type default `CHARACTER`.

`logical` is a rank-one allocatable array of type default `LOGICAL`.

`integer_pointer` is a rank-one pointer array of type `INTEGER(ip_)`.

`real_pointer` is a rank-one pointer array of type default `REAL(rp_)`

`complex_pointer` is a rank-one pointer array of type default `COMPLEX` (double precision complex in `GALAHAD_SLLS_-double`).

`character_pointer` is a rank-one pointer array of type default `CHARACTER`.

`logical_pointer` is a rank-one pointer array of type default `LOGICAL`.

All use is subject to the conditions of a BSD-3-Clause License.
See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

2.3.8 The derived data type for holding reverse-communication data

The derived data type `SLLS_reverse_type` is used to hold data needed for reverse communication when this is required. The components of `SLLS_reverse_type` are:

`nz_v_start` is a scalar variable of type `INTEGER(ip_)`, that may be used to hold the starting position in `NZ_v` (see below) of the list of indices of nonzero components of \mathbf{v} .

`nz_v_end` is a scalar variable of type `INTEGER(ip_)`, that may be used to hold the finishing position in `NZ_v` (see below) of the list of indices of nonzero components of \mathbf{v} .

`NZ_v` is a rank-one allocatable array of dimension n and type `INTEGER(ip_)`, that may be used to hold the indices of the nonzero components of \mathbf{v} . If used, components `NZ_v(nz_v_start:nz_v_end)` of \mathbf{V} (see below) will be nonzero.

\mathbf{V} is a rank-one allocatable array of dimension n and type `REAL(rp_)`, that is used to hold the components of the output vector \mathbf{v} .

\mathbf{P} is a rank-one allocatable array of dimension o and type `REAL(rp_)`, that is used to record the components of the resulting vector $\mathbf{A}_o \mathbf{v}$.

`nz_p_end` is a scalar variable of type `INTEGER(ip_)`, that is used to record the finishing position in `NZ_p` (see below) of the list of indices of nonzero components of $\mathbf{A}_o \mathbf{v}$ if required.

`NZ_p` is a rank-one allocatable array of dimension n and type `INTEGER(ip_)`, that is used to record the list of indices of nonzero components of $\mathbf{A}_o \mathbf{v}$ if required. Components `NZ_p(1:nz_prod_end)` of \mathbf{P} should then be nonzero.

2.4 Argument lists and calling sequences

There are three procedures for user calls (see Section 2.8 for further features):

1. The subroutine `SLLS_initialize` is used to set default values, and initialize private data, before solving one or more problems with the same sparsity and bound structure.
2. The subroutine `SLLS_solve` is called to solve the problem.
3. The subroutine `SLLS_terminate` is provided to allow the user to automatically deallocate array components of the private data, allocated by `SLLS_solve`, at the end of the solution process.

We use square brackets [] to indicate OPTIONAL arguments.

2.4.1 The initialization subroutine

Default values are provided as follows:

```
CALL SLLS_initialize( data, control )
```

`data` is a scalar `INTENT(INOUT)` argument of type `SLLS_data_type` (see Section 2.3.6). It is used to hold data about the problem being solved.

`control` is a scalar `INTENT(OUT)` argument of type `SLLS_control_type` (see Section 2.3.3). On exit, `control` contains default values for the components as described in Section 2.3.3. These values should only be changed after calling `SLLS_initialize`.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

2.4.2 The simplex-constrained linear least-squares subroutine

The simplex-constrained linear least-squares solution algorithm is called as follows:

```
CALL SLLS_solve( prob, X_stat, data, control, inform, userdata[, W, reverse, &
                eval_APROD, eval_ASPROD, eval_AFPROD, eval_PREC] )
```

`prob` is a scalar `INTENT(INOUT)` argument of type `QPT_problem_type` (see Section 2.3.2). It is used to hold data about the problem being solved. The user must allocate and set values for the array components, and set values for the component `prob%B`. Additionally, the user can provide A_o by allocating the relevant array components and setting values for `prob%Ao` using whichever of the matrix formats described in Section 2.1 is appropriate for the user's application; if the effect of A_o and its transpose are only available to form products via reverse communication (see `reverse` below) or with a set of user-supplied subroutines (see `eval_APROD` `eval_ASPROD` and `eval_AFPROD` below), `prob%Ao` is not needed.

The components `prob%X` must be set to initial estimates of the primal variables, \mathbf{x} . Inappropriate initial values will be altered, so the user should not be overly concerned if suitable values are not apparent, and may be content with merely setting `prob%X=0.0`. The components `prob%C`, `prob%G` and `prob%Z` need not be set or allocated on entry.

On exit, the components `prob%X` and `prob%Z` will contain the best estimates of the primal variables \mathbf{x} , and dual variables for the bound constraints \mathbf{z} , respectively. The components `prob%R` and `prob%G` will contain the residuals $\mathbf{r}(\mathbf{x})$ and gradients $\mathbf{g}(\mathbf{x})$ at \mathbf{x} , respectively. **Restrictions:** `prob%n` > 0, `prob%o` > 0 and (if A_o is provided) `prob%Ane` ≥ 0. `prob%Ao_type` ∈ {`'DENSE_BY_ROWS'`, `'DENSE_BY_COLUMNS'`, `'COORDINATE'`, `'SPARSE_BY_ROWS'`, `'SPARSE_BY_COLUMNS'`}.

`X_stat` is a rank-one `INTENT(INOUT)` array argument of dimension `prob%n` and type `INTEGER(ip_)`, that indicates which of the simple bound constraints are in the current working set. Possible values for `X_stat(j)`, $j = 1, \dots, \text{prob}\%n$, and their meanings are

- <0 the j -th simple bound constraint is in the working set, on its lower bound,
- >0 the j -th simple bound constraint is in the working set, on its upper bound, and
- 0 the j -th simple bound constraint is not in the working set.

Suitable values must be supplied if `control%sls_control%cold_start` = 0 on entry, but need not be provided for other input values of `control%cold_start`. Inappropriate values will be ignored. On exit, `X_stat` will contain values appropriate for the ultimate working set.

`data` is a scalar `INTENT(INOUT)` argument of type `SLLS_data_type` (see Section 2.3.6). It is used to hold data about the problem being solved. It must not have been altered **by the user** since the last call to `SLLS_initialize`.

`control` is a scalar `INTENT(IN)` argument of type `SLLS_control_type` (see Section 2.3.3). Default values may be assigned by calling `SLLS_initialize` prior to the first call to `SLLS_solve`.

`inform` is a scalar `INTENT(INOUT)` argument of type `SLLS_inform_type` (see Section 2.3.5). On initial entry, the component `status` must be set to 1, while other components need not be set. A successful call to `SLLS_solve` is indicated when the component `status` has the value 0. For other return values of `status`, see Sections 2.6 and 2.7.

`userdata` is a scalar `INTENT(INOUT)` argument of type `NLPT_userdata_type` whose components may be used to communicate user-supplied data (see Section 2.3.7) to and from the OPTIONAL subroutines `eval_APROD` and `eval_ASPROD` (see below).

`W` is an OPTIONAL rank-one `INTENT(INOUT)` array argument of dimension `prob%n` and type `REAL(rp_)` that if present contains the weights \mathbf{w} . If `W` is absent, weights of one will be used instead.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

`reverse` is an OPTIONAL scalar `INTENT(INOUT)` argument of type `SLLS_reverse_type` (see Section 2.3.8). It is used to communicate reverse-communication data between the subroutine and calling program. If `reverse` is PRESENT and `eval%APROD` or `eval%ASPROD` (see below) are absent, the user should monitor `inform%status` on exit (see Section 2.6).

`eval_APROD` is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the product $\mathbf{p} + \mathbf{A}_o \mathbf{v}$ or $\mathbf{p} + \mathbf{A}_o^T \mathbf{v}$ involving the Jacobian (and its transpose) and given vectors \mathbf{v} and \mathbf{p} . See Section 2.5.1 for details. If `eval_APROD` is present, it must be declared EXTERNAL in the calling program. If `eval_APROD` is absent, `GALAHAD_SLLS_solve` will use reverse communication (see Section 2.6) to obtain Jacobian-vector products if `reverse` is PRESENT or otherwise require that the user has provided all relevant components of `prob%A`.

`eval_ASPROD` is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the product $\mathbf{A}_o \mathbf{v}$ involving the Jacobian and a given *sparse* vector \mathbf{v} . See Section 2.5.2 for details. If `eval_ASPROD` is present, it must be declared EXTERNAL in the calling program. If `eval_ASPROD` is absent, `GALAHAD_SLLS_solve` will use reverse communication (see Section 2.6) to obtain Jacobian-sparse-vector products if `reverse` is PRESENT or otherwise require that the user has provided all relevant components of `prob%A`.

`eval_AFPROD` is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the product $\mathbf{A}_o \mathbf{v}$ or $\mathbf{A}_o^T \mathbf{v}$ involving the Jacobian (and its transpose) and a given vector \mathbf{v} in which either only some components of \mathbf{v} or the resulting product are set/required. See Section 2.5.3 for details. If `eval_AFPROD` is present, it must be declared EXTERNAL in the calling program. If `eval_AFPROD` is absent, `GALAHAD_SLLS_solve` will use reverse communication (see Section 2.6) to obtain Jacobian-sparse-vector products if `reverse` is PRESENT or otherwise require that the user has provided all relevant components of `prob%A`.

`eval_PREC` is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the product $\mathbf{P}^{-1} \mathbf{v}$ involving a symmetric, positive definite preconditioner \mathbf{P} and a given vector \mathbf{v} . See Section 2.5.4 for details. If `eval_PREC` is present, it must be declared EXTERNAL in the calling program. If `eval_PREC` is absent, `GALAHAD_SLLS_solve` will use reverse communication (see Section 2.6) to obtain preconditioning products so long as `reverse` is PRESENT; if `reverse` is not PRESENT, no preconditioning will be performed.

2.4.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

```
CALL SLLS_terminate( data, control, inform )
```

`data` is a scalar `INTENT(INOUT)` argument of type `SLLS_data_type` exactly as for `SLLS_solve`, which must not have been altered **by the user** since the last call to `SLLS_initialize`. On exit, array components will have been deallocated.

`control` is a scalar `INTENT(IN)` argument of type `SLLS_control_type` exactly as for `SLLS_solve`.

`inform` is a scalar `INTENT(OUT)` argument of type `SLLS_inform_type` exactly as for `SLLS_solve`. Only the component `status` will be set on exit, and a successful call to `SLLS_terminate` is indicated when this component status has the value 0. For other return values of `status`, see Section 2.7.

2.5 Matrix-vector operations

2.5.1 Jacobian-vector products via internal evaluation

If the argument `eval_APROD` is present when calling `GALAHAD_SLLS_solve`, the user is expected to provide a subroutine of that name to evaluate the sum $\mathbf{p} + \mathbf{A}_o \mathbf{v}$ or $\mathbf{p} + \mathbf{A}_o^T \mathbf{v}$ involving the product of the residual Jacobian \mathbf{A}_o or its transpose \mathbf{A}_o^T and a given vector \mathbf{v} . The routine must be specified as

```
SUBROUTINE eval_APROD( status, userdata, transpose, V, P )
```

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

whose arguments are as follows:

`status` is a scalar `INTENT (OUT)` argument of type `INTEGER (ip_)`, that should be set to 0 if the routine has been able to evaluate the sum $\mathbf{p} + \mathbf{A}_o \mathbf{v}$ or $\mathbf{p} + \mathbf{A}_o^T \mathbf{v}$ and to a non-zero value if the evaluation has not been possible.

`userdata` is a scalar `INTENT (INOUT)` argument of type `NLPT_userdata_type` whose components may be used to communicate user-supplied data to and from the subroutines `eval_APROD` (see Section 2.3.7).

`transpose` is a scalar `INTENT (IN)` array argument of type `default` that will be set `.TRUE.` if the product involves the transpose of the Jacobian \mathbf{A}_o^T and `.FALSE.` if the product involves the Jacobian \mathbf{A}_o itself.

`V` is a rank-one `INTENT (IN)` array argument of type `REAL (rp_)` whose components contain the vector \mathbf{v} .

`P` is a rank-one `INTENT (INOUT)` array argument of type `REAL (rp_)` whose components on input contain the vector \mathbf{p} and on output the sum $\mathbf{p} + \mathbf{A}_o \mathbf{v}$ when `%transpose` is `.FALSE.` or $\mathbf{p} + \mathbf{A}_o^T \mathbf{v}$ when `%transpose` is `.TRUE.`.

2.5.2 Jacobian-sparse-vector products via internal evaluation

If the argument `eval_ASPROD` is present when calling `GALAHAD_SLLS_solve`, the user is expected to provide a subroutine of that name to evaluate the product of the Jacobian \mathbf{A}_o with a given vector \mathbf{v} . The routine must be specified as

```
SUBROUTINE eval_ASPROD( status, userdata, V, P[, NZ_in, nz_in_start, nz_in_end,      &
                        NZ_out, nz_out_end] )
```

whose arguments are as follows:

`status` is a scalar `INTENT (OUT)` argument of type `INTEGER (ip_)`, that should be set to 0 if the routine has been able to evaluate the product $\mathbf{A}_o \mathbf{v}$ and to a non-zero value if the evaluation has not been possible.

`userdata` is a scalar `INTENT (INOUT)` argument of type `NLPT_userdata_type` whose components may be used to communicate user-supplied data to and from the subroutine (see Section 2.3.7).

`V` is a rank-one `INTENT (IN)` array argument of type `REAL (rp_)` whose components contain the vector \mathbf{v} . If components `nz_in_start`, `nz_in_end` and `NZ_in` (see below) are `PRESENT`, only components `NZ_in (nz_in_start:nz_in_end)` of `V` will be nonzero and the remaining components of `V` should be ignored. Otherwise, all components of `V` should be presumed to be nonzero.

`P` is a rank-one `INTENT (OUT)` array argument of type `REAL (rp_)` whose components on output contain the required components of $\mathbf{A}_o \mathbf{v}$. If components `nz_out_end` and `NZ_out` (see below) are `PRESENT`, only the nonzero components `NZ_out (1:nz_out_end)` of `P` need be assigned. Otherwise, all components of `P` must be set.

`nz_in_start` is an `OPTIONAL` scalar variable of type `INTEGER (ip_)`, that, if `PRESENT`, holds the starting position in `NZ_in` of the list of indices of nonzero components of \mathbf{v} .

`nz_in_end` is an `OPTIONAL` scalar variable of type `INTEGER (ip_)`, that, if `PRESENT`, holds the finishing position in `NZ_in` of the list of indices of nonzero components of \mathbf{v} .

`NZ_in` is an `OPTIONAL` rank-one array of dimension n and type `INTEGER (ip_)`, that, if `PRESENT`, holds the indices of the nonzero components of \mathbf{v} . If any of `nz_in_start`, `nz_in_end` and `NZ_in` are absent, all components of `V` are assumed to be nonzero.

`nz_out_end` is an `OPTIONAL` scalar variable of type `INTEGER (ip_)`, that, if `PRESENT`, must be set to record the number of non-zeros in $\mathbf{A}_o \mathbf{v}$.

`NZ_out` is an `OPTIONAL` rank-one array of dimension o and type `INTEGER (ip_)`, that, if `PRESENT`, must be set to record the list of indices of nonzero components of $\mathbf{A}_o \mathbf{v}$. If either of `nz_out_end` and `NZ_out` are absent, all components of `P` should be set even if they are zero.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

2.5.3 Jacobian-vector sub-products via internal evaluation

If the argument `eval_AFP` is present when calling `GALAHAD_SLLS_solve`, the user is expected to provide a subroutine of that name to evaluate the product of the Jacobian \mathbf{A}_o , or its transpose, with a given vector \mathbf{v} . Here, either only a subset of the components of the vector \mathbf{v} are nonzero, or only a subset of the components product $\mathbf{A}_o^T \mathbf{v}$ are to required. The routine must be specified as

```
SUBROUTINE eval_AFP( status, userdata, transpose, V, P, FREE, n_free )
```

whose arguments are as follows:

`status` is a scalar `INTENT(OUT)` argument of type `INTEGER(ip_)`, that should be set to 0 if the routine has been able to evaluate the sum $\mathbf{A}_o \mathbf{v}$ or $\mathbf{A}_o^T \mathbf{v}$ and to a non-zero value if the evaluation has not been possible.

`userdata` is a scalar `INTENT(INOUT)` argument of type `NLPT_userdata_type` whose components may be used to communicate user-supplied data to and from the subroutines `eval_APROD` (see Section 2.3.7).

`transpose` is a scalar `INTENT(IN)` array argument of type `default` that will be set `.TRUE.` if the product involves the transpose of the Jacobian \mathbf{A}_o^T and `.FALSE.` if the product involves the Jacobian \mathbf{A}_o itself.

`V` is a rank-one `INTENT(IN)` array argument of type `REAL(rp_)` whose components contain the vector \mathbf{v} . If `transpose` is `.FALSE.` only those components whose indices `FREE(:n_free)` (see below) will be set, and the remainder will be presumed to be zero.

`P` is a rank-one `INTENT(OUT)` array argument of type `REAL(rp_)` whose components on output must be set to the product $\mathbf{A}_o \mathbf{v}$ when `%transpose` is `.FALSE.`. If `%transpose` is `.TRUE.`, components with indices `FREE(:n_free)` (see below) should be set to the corresponding components of the product $\mathbf{A}_o^T \mathbf{v}$, and the remaining components ignored.

`FREE` is a rank-one `INTENT(IN)` array argument of type `INTEGER(ip_)` that flags the input components of \mathbf{v} that are set (when `transpose` is `.FALSE.`) or output components of $\mathbf{A}_o^T \mathbf{v}$ that are required (when `transpose` is `.TRUE.`). Specifically, only indices `FREE(:n_free)` of the relevant vector is set or required, and the remainder should be treated as zero (if `transpose` is `.FALSE.`) or ignored (if `transpose` is `.TRUE.`).

`n_free` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that specifies the number of components of `FREE` that need be considered.

2.5.4 Application of a preconditioner via internal evaluation

If the argument `eval_PREC` is present when calling `GALAHAD_SLLS_solve`, the user is expected to provide a subroutine of that name to evaluate the product $\mathbf{P}^{-1} \mathbf{v}$ involving a symmetric, positive definite preconditioner \mathbf{P} and a given n -vector \mathbf{v} . Ideally the n by n matrix \mathbf{P} should be chosen so that the of the product is inexpensive to compute, but also so that \mathbf{P} is a good approximation of $\mathbf{A}_o^T \mathbf{A}_o$ in the sense that the eigenvalues of $\mathbf{P}^{-1} \mathbf{A}_o^T \mathbf{A}_o$ are clustered around a small number of values (preferably all around 1.0). The routine must be specified as

```
SUBROUTINE eval_PREC( status, userdata, V, P )
```

whose arguments are as follows:

`status` is a scalar `INTENT(OUT)` argument of type `INTEGER(ip_)`, that should be set to 0 if the routine has been able to evaluate the product $\mathbf{P}^{-1} \mathbf{v}$, and to a non-zero value if the evaluation has not been possible.

`userdata` is a scalar `INTENT(INOUT)` argument of type `NLPT_userdata_type` whose components may be used to communicate user-supplied data to and from the subroutines `eval_APROD` (see Section 2.3.7).

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

- V is a rank-one `INTENT (IN)` array argument of type `REAL (rp_)` whose components contain the vector \mathbf{v} .
- P is a rank-one `INTENT (INOUT)` array argument of type `REAL (rp_)` whose components on output contain the vector $\mathbf{p} = \mathbf{P}^{-1}\mathbf{v}$.

2.6 Reverse Communication Information

A positive value of `inform%status` on exit from `SLLS_solve` indicates that `GALAHAD_SLLS_solve` is seeking further information—this will happen if the user has chosen to evaluate matrix-vector products by reverse communication. The user should compute the required information and re-enter `GALAHAD_SLLS_solve` with `inform%status` and all other arguments (except those specifically mentioned below) unchanged.

Possible values of `inform%status` and the information required are

2. The user should compute the product $\mathbf{A}_o\mathbf{v}$ involving the product of the residual Jacobian, \mathbf{A}_o , with a given vector \mathbf{v} . The vector \mathbf{v} is given in `reverse%V`, and the product $\mathbf{A}_o\mathbf{v}$ should be written to `reverse%P`, and `reverse%eval_status` should be set to 0. If the user is unable to evaluate the product, the user need not set `reverse%P`, but should then set `reverse%eval_status` to a non-zero value.
3. The user should compute the product $\mathbf{A}_o^T\mathbf{v}$ involving the product of the transpose of the residual Jacobian, \mathbf{A}_o , with a given vector \mathbf{v} . The vector \mathbf{v} is given in `reverse%V`, and the product $\mathbf{A}_o^T\mathbf{v}$ should be written to `reverse%P`, and `reverse%eval_status` should be set to 0. If the user is unable to evaluate the product, the user need not set `reverse%P`, but should then set `reverse%eval_status` to a non-zero value.
4. The user should compute the matrix-vector product $\mathbf{A}_o\mathbf{v}$ using the vector \mathbf{v} whose nonzero components are stored in positions `reverse%NZ_in(reverse%nz_in_start:reverse%nz_in_end)` of `reverse%V`. The remaining components of `reverse%V` should be ignored. The component `reverse%eval_status` should set to 0 unless the user is unable to evaluate the product, in which case `reverse%eval_status` should be set to a non-zero value, and the remaining components left unaltered.
5. The user should compute the nonzero components of the matrix-vector product $\mathbf{A}_o\mathbf{v}$ using the vector \mathbf{v} whose nonzero components are stored in positions `reverse%NZ_in(reverse%nz_in_start:reverse%nz_in_end)` of `reverse%V`. The remaining components of `reverse%V` should be ignored. The nonzero components must occupy positions `reverse%NZ_out(1:reverse%nz_out_end)` of `reverse%P`, and the components `reverse%NZ_out` and `reverse%nz_out_end` must be set. The component `reverse%eval_status` should set to 0 unless the user is unable to evaluate the product, in which case `reverse%eval_status` should be set to a non-zero value, and the remaining components left unaltered.
6. The user should compute the matrix-vector product $\mathbf{A}_o^T\mathbf{v}$ using the vector \mathbf{v} given in `reverse%V`. The components $(\mathbf{A}_o^T\mathbf{v})_j$ of the product $\mathbf{A}_o^T\mathbf{v}$, for $j = \text{reverse\%NZ_in}(\text{reverse\%nz_in_start}:\text{reverse\%nz_in_end})$, should be written to `reverse%P(j)`. The component `reverse%eval_status` should set to 0 unless the user is unable to evaluate the product, in which case `reverse%eval_status` should be set to a non-zero value, and the remaining components left unaltered.
7. The user should compute the product $\mathbf{P}^{-1}\mathbf{v}$ involving a symmetric, positive definite preconditioner \mathbf{P} and a given vector \mathbf{v} . The vector \mathbf{v} is given in `reverse%V`, and the product $\mathbf{P}^{-1}\mathbf{v}$ should be written to `reverse%P`, and `reverse%eval_status` should be set to 0. If the user is unable to evaluate the product, the user need not set `reverse%P`, but should then set `reverse%eval_status` to a non-zero value. This value of `inform%status` can only occur if the user has set `control%preconditioner = 2`.

2.7 Warning and error messages

A negative value of `inform%status` on exit from `SLLS_solve` or `SLLS_terminate` indicates that an error has occurred. No further calls should be made until the error has been corrected. Possible values are:

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

- 1. An allocation error occurred. A message indicating the offending array is written on unit `control%error`, and the returned allocation status and a string containing the name of the offending array are held in `inform%alloc-status` and `inform%bad_alloc` respectively.
- 2. A deallocation error occurred. A message indicating the offending array is written on unit `control%error` and the returned allocation status and a string containing the name of the offending array are held in `inform%alloc-status` and `inform%bad_alloc` respectively.
- 3. One of the restrictions `prob%n > 0`, `prob%o > 0` or the requirement that `prob%Ao-type` contain its relevant string 'DENSE_BY_ROWS', 'DENSE_BY_COLUMNS', 'COORDINATE', 'SPARSE_BY_ROWS' or 'SPARSE_BY_COLUMNS', when A_o is available, has been violated.
- 4. The bound constraints are inconsistent.
- 9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component `inform%factor-status`.
- 10. The factorization failed; the return status from the factorization package is given in the component `inform%factor-status`.
- 18. Too many iterations have been performed. This may happen if `control%maxit` is too small, but may also be symptomatic of a badly scaled problem.
- 19. The CPU time limit has been reached. This may happen if `control%cpu_time_limit` is too small, but may also be symptomatic of a badly scaled problem.

2.8 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable `control` of type `SLLS_control_type` (see Section 2.3.3), by reading an appropriate data specification file using the subroutine `SLLS_read_specfile`. This facility is useful as it allows a user to change SLLS control parameters without editing and recompiling programs that call SLLS.

A specification file, or *specfile*, is a data file containing a number of "specification commands". Each command occurs on a separate line, and comprises a "keyword", which is a string (in a close-to-natural language) used to identify a control parameter, and an (optional) "value", which defines the value to be assigned to the given control parameter. All keywords and values are case insensitive, keywords may be preceded by one or more blanks but values must not contain blanks, and each value must be separated from its keyword by at least one blank. Values must not contain more than 30 characters, and each line of the specfile is limited to 80 characters, including the blanks separating keyword and value.

The portion of the specification file used by `SLLS_read_specfile` must start with a "BEGIN SLLS" command and end with an "END" command. The syntax of the specfile is thus defined as follows:

```
( .. lines ignored by SLLS_read_specfile .. )
BEGIN SLLS
  keyword      value
  .....
  keyword      value
END
( .. lines ignored by SLLS_read_specfile .. )
```

where keyword and value are two strings separated by (at least) one blank. The "BEGIN SLLS" and "END" delimiter command lines may contain additional (trailing) strings so long as such strings are separated by one or more blanks, so that lines such as

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

```
BEGIN SLLS SPECIFICATION
```

and

```
END SLLS SPECIFICATION
```

are acceptable. Furthermore, between the “BEGIN SLLS” and “END” delimiters, specification commands may occur in any order. Blank lines and lines whose first non-blank character is ! or * are ignored. The content of a line after a ! or * character is also ignored (as is the ! or * character itself). This provides an easy manner to “comment out” some specification commands, or to comment specific values of certain control parameters.

The value of a control parameters may be of three different types, namely integer, logical or real. Integer and real values may be expressed in any relevant Fortran integer and floating-point formats (respectively). Permitted values for logical parameters are “ON”, “TRUE”, “.TRUE.”, “T”, “YES”, “Y”, or “OFF”, “NO”, “N”, “FALSE”, “.FALSE.” and “F”. Empty values are also allowed for logical control parameters, and are interpreted as “TRUE”.

The specification file must be open for input when `SLLS_read_specfile` is called, and the associated device number passed to the routine in `device` (see below). Note that the corresponding file is `REWINDED`, which makes it possible to combine the specifications for more than one program/routine. For the same reason, the file is not closed by `SLLS_read_specfile`.

2.8.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL SLLS_read_specfile( control, device )
```

`control` is a scalar `INTENT(INOUT)` argument of type `SLLS_control_type` (see Section 2.3.3). Default values should have already been set, perhaps by calling `SLLS_initialize`. On exit, individual components of `control` may have been changed according to the commands found in the specfile. Specfile commands and the component (see Section 2.3.3) of `control` that each affects are given in Table 2.1.

`device` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that must be set to the unit number on which the specfile has been opened. If `device` is not open, `control` will not be altered and execution will continue, but an error message will be printed on unit `control%error`.

2.9 Information printed

If `control%print_level` is positive, information about the progress of the algorithm will be printed on unit `control%out`. If `control%print_level = 1`, a single line of output will be produced for each iteration of the process. This will give the total number of CG iterations performed (if any), the value of the objective function and the norm of the projected gradient, the stepsize taken, the number of free variables (i.e., those that have a positive value), the change in the number of free variables since the last iteration, and the elapsed CPU time in seconds. If `control%print_level ≥ 2` this output will be increased to provide significant detail of each iteration. This extra output includes residuals of the linear systems solved, and, for larger values of `control%print_level`, values of the primal and dual variables and Lagrange multiplier.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

command	component of control	value type
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
start-print	%start_print	integer
stop-print	%stop_print	integer
iterations-between-printing	%print_gap	integer
maximum-number-of-iterations	%maxit	integer
cold-start	%cold_start	integer
preconditioner	%preconditioner	integer
max-change-to-working-set-for-subspace-solution	%change_max	integer
maximum-number-of-cg-iterations-per-iteration	%cg_maxit	integer
maximum-number-of-arcsearch-steps	%arcsearch_max_steps	integer
primal-accuracy-required	%stop_p	real
dual-accuracy-required	%stop_d	real
complementary-slackness-accuracy-required	%stop_c	real
cg-relative-accuracy-required	%stop_cg_relative	real
cg-absolute-accuracy-required	%stop_cg_absolute	real
maximum-arcsearch-stepsize	%alpha_max	real
initial-arcsearch-stepsize	%alpha_initial	real
arcsearch-reduction-factor	%alpha_reduction	real
arcsearch-acceptance-tolerance	%arcsearch_acceptance_tol	real
regularization-weight	%regularization_weight	real
maximum-cpu-time-limit	%cpu_time_limit	real
direct-subproblem-solve	%direct_subproblem_solve	logical
exact-arc-search-used	%exact_arc_search	logical
space-critical	%space_critical	logical
deallocate-error-fatal	%deallocate_error_fatal	logical
output-line-prefix	%prefix	character

Table 2.1: Specfile commands and associated components of control.

Other modules used directly: SLLS_solve calls the GALAHAD packages GALAHAD_CPU_time, GALAHAD_SYMBOLS, GALAHAD_SPACE, GALAHAD_STRING, GALAHAD_SORT, GALAHAD_NORMS, GALAHAD_CONVERT, GALAHAD_SBLS, GALAHAD_QPT, GALAHAD_QPD, GALAHAD_USERDATA and GALAHAD_SPECFILE.

Input/output: Output is under control of the arguments control%error, control%out and control%print_level.

Restrictions: $\text{prob}\%n > 0$, $\text{prob}\%o > 0$, $\text{prob}\%A_type \in \{ 'DENSE_BY_ROWS', 'DENSE_BY_COLUMNS', 'COORDINATE', 'SPARSE_BY_ROWS', 'SPARSE_BY_COLUMNS' \}$ (if \mathbf{A}_o is explicit).

Portability: ISO Fortran 95 + TR 15581 or Fortran 2003. The package is thread-safe.

4 METHOD

The required solution \mathbf{x} necessarily satisfies the primal optimality conditions

$$\mathbf{e}^T \mathbf{x} = 1 \text{ and } \mathbf{x} \geq 0, \quad (4.1)$$

the dual optimality conditions

$$\mathbf{A}_o^T \mathbf{W}(\mathbf{A}_o \mathbf{x} - \mathbf{b}) + \sigma \mathbf{x} = \lambda \mathbf{e} + \mathbf{z}, \quad (4.2)$$

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

for some scalar Lagrange multiplier λ and dual variables

$$\mathbf{z} \geq 0, \quad (4.3)$$

and the complementary slackness conditions

$$\mathbf{x}^T \mathbf{z} = 0, \quad (4.4)$$

where the vector inequalities hold componentwise. Projected-gradient methods iterate towards a point that satisfies these conditions by ultimately aiming to satisfy (4.2), while ensuring that (4.1), and (4.3) and (4.4) are satisfied at each stage. Appropriate norms of the amounts by which (4.1), (4.2) and (4.4) fail to be satisfied are known as the primal and dual infeasibility, and the violation of complementary slackness, respectively.

The method is iterative. Each iteration proceeds in two stages. Firstly, a search direction \mathbf{s} from the current estimate of the solution \mathbf{x} is computed. This may be in a scaled steepest-descent direction, or, if the working set of variables on bounds has not changed dramatically, in a direction that provides an approximate minimizer of the objective over a subspace comprising the currently free-variables. The latter is computed either using an appropriate sparse factorization by the package `GALAHAD_SBLs`, or by the conjugate-gradient least-squares (CGLS) method; it may be necessary to regularize the subproblem very slightly to avoid a ill-posedness. Thereafter, a piecewise linesearch (arc search) is carried out along the arc $\mathbf{x}(\alpha) = P(\mathbf{x} + \alpha \mathbf{s})$ for $\alpha > 0$, where the projection operator $P(\mathbf{v})$ gives the nearest point to \mathbf{v} within the regular simplex; thus this arc bends the search direction into the feasible region. The arc search is performed either exactly, by passing through a set of increasing breakpoints at which it changes direction, or inexactly, by evaluating a sequence of different α on the arc. All computation is designed to exploit sparsity in \mathbf{A}_o .

References:

Full details are provided in

N. I. M. Gould (2022). Linear least-squares over the unit simplex. In preparation.

5 EXAMPLE OF USE

Suppose we wish to minimize

$$\frac{1}{2} \left\| \begin{pmatrix} x_1 \\ x_1 + x_2 - 2 \\ x_3 - 1 \\ x_3 - 2 \end{pmatrix} \right\|_2^2$$

within the simplex

$$x_1 + x_2 + x_3 = 1, \quad x_1, x_2, x_3 \geq 0.$$

Then, on writing the data for this problem as

$$\mathbf{A}_o = \begin{pmatrix} 1 & & & \\ 1 & 1 & & \\ & & 1 & \\ & & & 1 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 0 \\ 2 \\ 1 \\ 2 \end{pmatrix}, \quad \mathbf{x}' = \begin{pmatrix} -1 \\ -\infty \\ 0 \end{pmatrix} \quad \text{and} \quad \mathbf{x}'' = \begin{pmatrix} \infty \\ 1 \\ 2 \end{pmatrix}$$

in sparse coordinate format, we may use the following code:

```
! THIS VERSION: GALAHAD 4.3 - 2023-12-31 AT 10:15 GMT
PROGRAM GALAHAD_SLLS_EXAMPLE
USE GALAHAD_SLLS_double      ! double precision version
IMPLICIT NONE
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
TYPE ( QPT_problem_type ) :: p
```

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

```

TYPE ( SLLS_data_type ) :: data
TYPE ( SLLS_control_type ) :: control
TYPE ( SLLS_inform_type ) :: inform
TYPE ( GALAHAD_userdata_type ) :: userdata
INTEGER, ALLOCATABLE, DIMENSION( : ) :: X_stat
INTEGER :: s
INTEGER, PARAMETER :: n = 3, o = 4, a_ne = 5
! start problem data
ALLOCATE( p%B( o ), p%X( n ), X_stat( n ) )
p%n = n ; p%o = o ! dimensions
p%B = (/ 0.0_wp, 2.0_wp, 1.0_wp, 2.0_wp /) ! right-hand side
p%X = 0.0_wp ! start from zero
! sparse co-ordinate storage format
CALL SMT_put( p%Ao%type, 'COORDINATE', s ) ! Co-ordinate storage for A
ALLOCATE( p%Ao%val( a_ne ), p%Ao%row( a_ne ), p%Ao%col( A_ne ) )
p%Ao%m = o ; p%Ao%n = n
p%Ao%val = (/ 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /) ! Jacobian A
p%Ao%row = (/ 1, 2, 2, 3, 4 /) !
p%Ao%col = (/ 1, 1, 2, 3, 3 /) ; p%Ao%ne = a_ne
! problem data complete
CALL SLLS_initialize( data, control, inform ) ! Initialize control parameters
! control%print_level = 1 ! print one line/iteration
control%exact_arc_search = .FALSE.
! control%CONVERT_control%print_level = 3
inform%status = 1
CALL SLLS_solve( p, X_stat, data, control, inform, userdata )
IF ( inform%status == 0 ) THEN ! Successful return
  WRITE( 6, "( /, ' SLLS: ', I0, ' iterations ', /,
    & ' Optimal objective value =',
    & ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )
    inform%iter, inform%obj, p%X
  ELSE ! Error returns
    WRITE( 6, "( /, ' SLLS_solve exit status = ', I0 )" ) inform%status
    WRITE( 6, * ) inform%alloc_status, inform%bad_alloc
  END IF
CALL SLLS_terminate( data, control, inform ) ! delete workspace
DEALLOCATE( p%B, p%X, p%Z, p%R, p%G, X_stat )
DEALLOCATE( p%Ao%val, p%Ao%row, p%Ao%col, p%Ao%type )
END PROGRAM GALAHAD_SLLS_EXAMPLE

```

This produces the following output:

```

SLLS: 5 iterations
Optimal objective value = 2.3333E+00
Optimal solution = 3.3346E-13 3.3333E-01 6.6667E-01

```

The same problem may be solved holding the data in a sparse row-wise storage format by replacing the lines

```

! sparse coordinate storage format
...
! problem data complete

```

by

```

! sparse row-wise storage format
CALL SMT_put( p%A%type, 'SPARSE_BY_ROWS', s ) ! Specify sparse-by-rows
ALLOCATE( p%A%val( a_ne ), p%A%col( a_ne ), p%A%ptr( m + 1 ) )

```

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

```
p%A%val = (/ 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /) ! Jacobian A
p%A%col = (/ 1, 1, 2, 3, 3 /) ! Column indices
p%A%ptr = (/ 1, 2, 4, 5, 6 /) ! Set row pointers
! problem data complete
```

a sparse column-wise storage format by replacing the lines

```
! sparse column-wise storage format
CALL SMT_put( p%A%type, 'SPARSE_BY_COLUMNS', s ) ! Specify sparse-by-columns
ALLOCATE( p%A%val( a_ne ), p%A%row( a_ne ), p%A%ptr( n + 1 ) )
p%A%val = (/ 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /) ! Jacobian A
p%A%row = (/ 1, 2, 2, 3, 4 /) ! Row indices
p%A%ptr = (/ 1, 3, 4, 6 /) ! Set column pointers
! problem data complete
```

a dense-by-rows storage format with the replacement lines

```
! dense storage format
CALL SMT_put( p%A%type, 'DENSE_BY_ROWS', s ) ! Specify dense-by-rows
ALLOCATE( p%A%val( m * n ) )
p%A%val = (/ 1.0_wp, 0.0_wp, 0.0_wp, 1.0_wp, 1.0_wp, 0.0_wp, 0.0_wp, &
            1.0_wp, 0.0_wp, 0.0_wp, 1.0_wp /)
! problem data complete
```

or a dense-by-columns storage format using the replacement

```
! dense storage format
CALL SMT_put( p%A%type, 'DENSE_BY_COLUMNS', s ) ! Specify dense-by-columns
ALLOCATE( p%A%val( m * n ) )
p%A%val = (/ 1.0_wp, 1.0_wp, 0.0_wp, 0.0_wp, 0.0_wp, 1.0_wp, 0.0_wp, &
            0.0_wp, 0.0_wp, 1.0_wp, 1.0_wp /)
! problem data complete
```

respectively.

The same problem may be solved using reverse communication with the following code:

```
! THIS VERSION: GALAHAD 4.3 - 2022-12-31 AT 11:00 GMT.
PROGRAM GALAHAD_SLLS_SECOND_EXAMPLE ! reverse communication interface
USE GALAHAD_SLLS_double ! double precision version
IMPLICIT NONE
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
TYPE ( QPT_problem_type ) :: p
TYPE ( SLLS_data_type ) :: data
TYPE ( SLLS_control_type ) :: control
TYPE ( SLLS_inform_type ) :: inform
TYPE ( SLLS_reverse_type ) :: reverse
TYPE ( GALAHAD_userdata_type ) :: userdata
INTEGER, ALLOCATABLE, DIMENSION( : ) :: X_stat
INTEGER :: i, j, k, l, nflag
REAL ( KIND = wp ) :: val
INTEGER, PARAMETER :: n = 3, o = 4, Ao_ne = 5
INTEGER, ALLOCATABLE, DIMENSION( : ) :: Ao_row, Ao_ptr, FLAG
REAL ( KIND = wp ), ALLOCATABLE, DIMENSION( : ) :: Ao_val
! start problem data
ALLOCATE( p%B( o ), p%X( n ), X_stat( n ) )
p%n = n ; p%o = o ! dimensions
```

All use is subject to the conditions of a BSD-3-Clause License.
See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

```

p%B = (/ 0.0_wp, 2.0_wp, 1.0_wp, 2.0_wp /) ! right-hand side
p%X = 0.0_wp ! start from zero
! sparse column storage format
ALLOCATE( Ao_val( Ao_ne ), Ao_row( Ao_ne ), Ao_ptr( n + 1 ) )
Ao_val = (/ 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /) ! Jacobian A by columns
Ao_row = (/ 1, 2, 2, 3, 4 /) ! row indices
Ao_ptr = (/ 1, 3, 4, 6 /) ! pointers to column starts
! problem data complete
CALL SLLS_initialize( data, control, inform ) ! Initialize control parameters
! control%print_level = 1 ! print one line/iteration
control%exact_arc_search = .FALSE.
ALLOCATE( FLAG( n ) )
nflag = 0 ; FLAG = 0 ! Flag if index already used in current (nflag) product
inform%status = 1
10 CONTINUE ! Solve problem - reverse communication loop
CALL SLLS_solve( p, X_stat, data, control, inform, userdata, &
reverse = reverse )
SELECT CASE ( inform%status )
CASE ( 0 ) ! successful return
WRITE( 6, "( /, ' SLLS: ', I0, ' iterations ', /, &
& ' Optimal objective value = ', &
& ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" ) &
inform%iter, inform%obj, p%X
CASE ( 2 ) ! compute A * v
reverse%P( : o ) = 0.0_wp
DO j = 1, n
val = reverse%V( j )
DO k = Ao_ptr( j ), Ao_ptr( j + 1 ) - 1
i = Ao_row( k )
reverse%P( i ) = reverse%P( i ) + Ao_val( k ) * val
END DO
END DO
GO TO 10
CASE ( 3 ) ! compute A^T * v
reverse%P( : n ) = 0.0_wp
DO j = 1, n
val = 0.0_wp
DO k = Ao_ptr( j ), Ao_ptr( j + 1 ) - 1
val = val + Ao_val( k ) * reverse%V( Ao_row( k ) )
END DO
reverse%P( j ) = val
END DO
GO TO 10
CASE ( 4 ) ! compute A * sparse v
reverse%P( : o ) = 0.0_wp
DO l = reverse%nz_in_start, reverse%nz_in_end
j = reverse%NZ_in( l )
val = reverse%V( j )
DO k = Ao_ptr( j ), Ao_ptr( j + 1 ) - 1
i = Ao_row( k )
reverse%P( i ) = reverse%P( i ) + Ao_val( k ) * val
END DO
END DO
GO TO 10
CASE ( 5 ) ! compute sparse( A * sparse v )

```

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

```

nflag = nflag + 1
reverse%nz_out_end = 0
DO l = reverse%nz_in_start, reverse%nz_in_end
  j = reverse%NZ_in( l )
  val = reverse%V( j )
  DO k = Ao_ptr( j ), Ao_ptr( j + 1 ) - 1
    i = Ao_row( k )
    IF ( FLAG( i ) < nflag ) THEN
      FLAG( i ) = nflag
      reverse%P( i ) = Ao_val( k ) * val
      reverse%nz_out_end = reverse%nz_out_end + 1
      reverse%NZ_out( reverse%nz_out_end ) = i
    ELSE
      reverse%P( i ) = reverse%P( i ) + Ao_val( k ) * val
    END IF
  END DO
END DO
GO TO 10
CASE ( 6 ) ! compute sparse( AT * v )
  reverse%P( : n ) = 0.0_wp
  DO l = reverse%nz_in_start, reverse%nz_in_end
    j = reverse%NZ_in( l )
    val = 0.0_wp
    DO k = Ao_ptr( j ), Ao_ptr( j + 1 ) - 1
      val = val + Ao_val( k ) * reverse%V( Ao_row( k ) )
    END DO
    reverse%P( j ) = val
  END DO
GO TO 10
CASE DEFAULT ! error returns
  WRITE( 6, "( /, ' SLLS_solve exit status = ', I0 ) " ) inform%status
END SELECT
CALL SLLS_terminate( data, control, inform ) ! delete workspace
DEALLOCATE( p%B, p%X, p%Z, X_stat, FLAG )
DEALLOCATE( Ao_val, Ao_row, Ao_ptr )
END PROGRAM GALAHAD_SLLS_SECOND_EXAMPLE

```

This produces the following output:

```

SLLS: 5 iterations
Optimal objective value = 2.3333E+00
Optimal solution = 4.7184E-16 3.3333E-01 6.6667E-01

```

The same problem may also be solved by user-provided matrix-vector products as follows:

```

! THIS VERSION: GALAHAD 4.3 - 2022-12-30 AT 16:20 GMT.
PROGRAM GALAHAD_SLLS_THIRD_EXAMPLE ! subroutine evaluation interface
USE GALAHAD_SLLS_double           ! double precision version
IMPLICIT NONE
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
TYPE ( QPT_problem_type ) :: p
TYPE ( SLLS_data_type ) :: data
TYPE ( SLLS_control_type ) :: control
TYPE ( SLLS_inform_type ) :: inform
TYPE ( GALAHAD_userdata_type ) :: userdata
INTEGER, ALLOCATABLE, DIMENSION( : ) :: X_stat

```

All use is subject to the conditions of a BSD-3-Clause License.
See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

```

    INTEGER, PARAMETER :: n = 3, o = 4, Ao_ne = 5
! partition userdata%integer so that it holds
!   o n nflag  flag      Ao_ptr      Ao_row
!   |1|2| 3  |4 to n+3 |n+4 to 2n+4|2n+5 to 2n+4+a_ne|
! partition userdata%real so that it holds
!   Ao_val
!   |1 to Ao_ne|
    INTEGER, PARAMETER :: mn = MAX( o, n )
    INTEGER, PARAMETER :: nflag = 3, st_flag = 3, st_ptr = st_flag + mn
    INTEGER, PARAMETER :: st_row = st_ptr + n + 1, st_val = 0
    INTEGER, PARAMETER :: len_integer = st_row + Ao_ne + 1, len_real = Ao_ne
    EXTERNAL :: APROD, ASPROD, AFPROD
! start problem data
    ALLOCATE( p%B( o ), p%X( n ), X_stat( n ) )
    p%n = n ; p%o = o ! dimensions
    p%B = (/ 0.0_wp, 2.0_wp, 1.0_wp, 2.0_wp /) ! right-hand side
    p%X = 0.0_wp ! start from zero
! sparse co-ordinate storage format
    ALLOCATE( userdata%integer( len_integer ), userdata%real( len_real ) )
    userdata%integer( 1 ) = o ! load Jacobian data into userdata
    userdata%integer( 2 ) = n
    userdata%integer( st_ptr + 1 : st_ptr + n + 1 ) = (/ 1, 3, 4, 6 /)
    userdata%integer( st_row + 1 : st_row + Ao_ne ) = (/ 1, 2, 2, 3, 4 /)
    userdata%real( st_val + 1 : st_val + Ao_ne ) = (/ 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /) &
! problem data complete
    CALL SLLS_initialize( data, control, inform ) ! Initialize control parameters
!   control%print_level = 1 ! print one line/iteration
!   control%exact_arc_search = .FALSE.
! load workspace into userdata
    userdata%integer( nflag ) = 0
    userdata%integer( st_flag + 1 : st_flag + mn ) = 0
    inform%status = 1
    CALL SLLS_solve( p, X_stat, data, control, inform, userdata, &
                    eval_APROD = APROD, eval_ASPROD = ASPROD, &
                    eval_AFPROD = AFPROD )
    IF ( inform%status == 0 ) THEN ! Successful return
        WRITE( 6, "( /, ' SLLS: ', I0, ' iterations ', /, &
                & ' Optimal objective value =', &
                & ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" ) &
        inform%iter, inform%obj, p%X
    ELSE ! Error returns
        WRITE( 6, "( /, ' SLLS_solve exit status = ', I0 ) " ) inform%status
    END IF
    CALL SLLS_terminate( data, control, inform ) ! delete workspace
    DEALLOCATE( p%B, p%X, p%Z, X_stat )
    DEALLOCATE( userdata%integer, userdata%real )
END PROGRAM GALAHAD_SLLS_THIRD_EXAMPLE

SUBROUTINE APROD( status, userdata, transpose, V, P )
    USE GALAHAD_USERDATA_double
    INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
    INTEGER, INTENT( OUT ) :: status
    TYPE ( GALAHAD_userdata_type ), INTENT( INOUT ) :: userdata
    LOGICAL, INTENT( IN ) :: transpose

```

All use is subject to the conditions of a BSD-3-Clause License.
See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

```
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: V
REAL ( KIND = wp ), DIMENSION( : ), INTENT( INOUT ) :: P
INTEGER :: i, j, k
REAL ( KIND = wp ) :: val
! recover problem data from userdata
INTEGER :: o, n, nflag, st_flag, st_ptr, st_row, st_val
o = userdata%integer( 1 )
n = userdata%integer( 2 )
nflag = 3
st_flag = 3
st_ptr = st_flag + MAX( o, n )
st_row = st_ptr + n + 1
st_val = 0
IF ( transpose ) THEN
  DO j = 1, n
    DO k = userdata%integer( st_ptr + j ),
      userdata%integer( st_ptr + j + 1 ) - 1
      P( j ) = P( j ) + userdata%real( st_val + k ) *
        V( userdata%integer( st_row + k ) )
    END DO
  END DO
ELSE
  DO j = 1, n
    val = V( j )
    DO k = userdata%integer( st_ptr + j ),
      userdata%integer( st_ptr + j + 1 ) - 1
      i = userdata%integer( st_row + k )
      P( i ) = P( i ) + userdata%real( st_val + k ) * val
    END DO
  END DO
END IF
status = 0
RETURN
END SUBROUTINE APROD

SUBROUTINE ASPROD( status, userdata, V, P, NZ_in, nz_in_start, nz_in_end,
  NZ_out, nz_out_end )
USE GALAHAD_USERDATA_double
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
TYPE ( GALAHAD_userdata_type ), INTENT( INOUT ) :: userdata
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: V
REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: P
INTEGER, OPTIONAL, INTENT( IN ) :: nz_in_start, nz_in_end
INTEGER, OPTIONAL, INTENT( INOUT ) :: nz_out_end
INTEGER, DIMENSION( : ), OPTIONAL, INTENT( IN ) :: NZ_in
INTEGER, DIMENSION( : ), OPTIONAL, INTENT( INOUT ) :: NZ_out
INTEGER :: i, j, k, l
REAL ( KIND = wp ) :: val
! recover problem data from userdata
INTEGER :: o, n, nflag, st_flag, st_ptr, st_row, st_val
IF ( PRESENT( NZ_in ) ) THEN
  IF ( .NOT. ( PRESENT( nz_in_start ) .AND. PRESENT( nz_in_end ) ) ) THEN
    status = - 1 ; RETURN
  END IF
```

All use is subject to the conditions of a BSD-3-Clause License.
See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

```

END IF
o = userdata%integer( 1 )
n = userdata%integer( 2 )
nflag = 3
st_flag = 3
st_ptr = st_flag + MAX( o, n )
st_row = st_ptr + n + 1
st_val = 0
IF ( PRESENT( NZ_in ) ) THEN
  IF ( PRESENT( NZ_out ) ) THEN
    IF ( .NOT. PRESENT( nz_out_end ) ) THEN
      status = - 1 ; RETURN
    END IF
    userdata%integer( nflag ) = userdata%integer( nflag ) + 1
    nz_out_end = 0
    DO l = nz_in_start, nz_in_end
      j = NZ_in( l )
      val = V( j )
      DO k = userdata%integer( st_ptr + j ),
        userdata%integer( st_ptr + j + 1 ) - 1
        i = userdata%integer( st_row + k )
        IF ( userdata%integer( st_flag + i ) <
          userdata%integer( nflag ) ) THEN
          userdata%integer( st_flag + i ) = userdata%integer( nflag )
          P( i ) = userdata%real( st_val + k ) * val
          nz_out_end = nz_out_end + 1
          NZ_out( nz_out_end ) = i
        ELSE
          P( i ) = P( i ) + userdata%real( st_val + k ) * val
        END IF
      END DO
    END DO
  ELSE
    P( : o ) = 0.0_wp
    DO l = nz_in_start, nz_in_end
      j = NZ_in( l )
      val = V( j )
      DO k = userdata%integer( st_ptr + j ),
        userdata%integer( st_ptr + j + 1 ) - 1
        i = userdata%integer( st_row + k )
        P( i ) = P( i ) + userdata%real( st_val + k ) * val
      END DO
    END DO
  END IF
ELSE
  IF ( PRESENT( NZ_out ) ) THEN
    IF ( .NOT. PRESENT( nz_out_end ) ) THEN
      status = - 1 ; RETURN
    END IF
    userdata%integer( nflag ) = userdata%integer( nflag ) + 1
    nz_out_end = 0
    DO j = 1, n
      val = V( j )
      DO k = userdata%integer( st_ptr + j ),
        userdata%integer( st_ptr + j + 1 ) - 1

```

All use is subject to the conditions of a BSD-3-Clause License.
See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

```

        i = userdata%integer( st_row + k )
        IF ( userdata%integer( st_flag + i ) <
            userdata%integer( nflag ) ) THEN
            userdata%integer( st_flag + i ) = userdata%integer( nflag )
            P( i ) = userdata%real( st_val + k ) * val
            nz_out_end = nz_out_end + 1
            NZ_out( nz_out_end ) = i
        ELSE
            P( i ) = P( i ) + userdata%real( st_val + k ) * val
        END IF
    END DO
END DO
ELSE
    P( : o ) = 0.0_wp
    DO j = 1, n
        val = V( j )
        DO k = userdata%integer( st_ptr + j ),
            userdata%integer( st_ptr + j + 1 ) - 1
            i = userdata%integer( st_row + k )
            P( i ) = P( i ) + userdata%real( st_val + k ) * val
        END DO
    END DO
END IF
END IF
status = 0
RETURN
END SUBROUTINE ASPROD

SUBROUTINE AFPROD( status, userdata, transpose, V, P, FREE, n_free )
USE GALAHAD_USERDATA_double
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
TYPE ( GALAHAD_userdata_type ), INTENT( INOUT ) :: userdata
LOGICAL, INTENT( IN ) :: transpose
INTEGER, INTENT( IN ) :: n_free
INTEGER, INTENT( IN ), DIMENSION( : ) :: FREE
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: V
REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: P
INTEGER :: i, j, k, l
REAL ( KIND = wp ) :: val
! recover problem data from userdata
INTEGER :: o, n, nflag, st_flag, st_ptr, st_row, st_val
o = userdata%integer( 1 )
n = userdata%integer( 2 )
nflag = 3
st_flag = 3
st_ptr = st_flag + MAX( o, n )
st_row = st_ptr + n + 1
st_val = 0
IF ( transpose ) THEN
    DO l = 1, n_free
        j = FREE( l )
        val = 0.0_wp
        DO k = userdata%integer( st_ptr + j ),
            userdata%integer( st_ptr + j + 1 ) - 1

```

All use is subject to the conditions of a BSD-3-Clause License.
See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

```

        val = val + userdata%real( st_val + k ) *
            V( userdata%integer( st_row + k ) )
    END DO
    P( j ) = val
END DO
ELSE
    P( : o ) = 0.0_wp
    DO l = 1, n_free
        j = FREE( l )
        val = V( j )
        DO k = userdata%integer( st_ptr + j ),
            userdata%integer( st_ptr + j + 1 ) - 1
            i = userdata%integer( st_row + k )
            P( i ) = P( i ) + userdata%real( st_val + k ) * val
        END DO
    END DO
END IF
status = 0
RETURN
END SUBROUTINE AFFPROD

```

This produces the same output. Now notice how the matrix \mathbf{A}_o is passed to the matrix-vector product evaluation routines via the integer and real components of the derived type `userdata`.